Lab 4: Mental Binary Math Game with ROM-based Game Access Control on FPGA

Mubashar Khan

ECE5440

# Introduction

In this lab, we add ROM capabilities to the design of the single-player binary mental math game using the DE2-115 FPGA board. Before the player can play the game, they must first enter a 16-bit password which has been predefined. User inputs will only be accepted once the user presses the corresponding button. The password is checked by comparing it to the password defined in ROM. Once the player has entered the password, they can specify the time they want the game to last up to 99 seconds. The game functions in a round-based system with FPGA generating a random 4-bit binary number once the random button is pressed. That number is then displayed on a 7-segment display which can display the terms 0-F in hexadecimal. The player can look at the number being displayed on the 7-segment display and mentally should convert it to binary. Then the player should enter a number using their 4-bit toggle switches such that the sum of FPGA’s random number and the player’s 4-bit binary numbers add to 1111 in binary. The player’s 4-bit binary input will also be displayed using a second 7-segment display. The FPGA will compute the sum of both numbers and display the sum on a third 7-segment display. If the third 7-segment display shows ‘F’, signifying 1111 in hexadecimal, the player wins the round and gains 1 point.

# System Architecture Design

The diagram in figure 1 shows the top-level system architecture for the binary mental math game with access control features along with a hardware based true random number generator using timer control. To begin, the player will have to enter a password to gain access to the FPGA. The password for boards using the code in this document is 2949. Each decimal digit must be entered as a binary number and then the player must press the password button shown in figure 1. When pushed, the button sends a signal to the button shaper which outputs a pulse that is one clock cycle long. This process is repeated for each digit of the password. If players enter an incorrect password, the program will reset after every 4 inputs until the correct password is entered. Another option is to press the reset button. The board will determine if the entered password is valid by comparing the entered password to the value stored in ROM.

Once the player has entered a valid password, they must choose the time limit for the game.

The player will have a set of 4 toggle switches to input their 4-bit binary numbers. To set the time, they will need to press the password button to enter the time setting state and then press the password button. Pressing the password button a third time will begin the game and turn the countdown timer on. When a player inputs a value, they must press the load button to load it into the FPGA. The input is sent to the load register. If the correct password has been entered, the access controller will allow the load button signal through to the load register. This will take the input and send it to a 7-segment display and to the adder.

The game uses a 7-segment display which displays a number based on a 4-bit binary input. This is accomplished using a 4-16 decoder. The decoder takes the 4-bit input and converts it to a hexadecimal value. That value is then displayed on a 7-segment display. When both players have entered a value into the FPGA, the sum is found using a binary adder. The adder takes two 4-bit inputs and outputs a 4-bit value. We do not care about the overflow bit and any sums found that are over binary 1111 will wrap around and start counting from binary 0000. The output from the adder is passed onto the 4-16 decoder which converts the sum to a hexadecimal value that can be displayed on a 7-segment display.

FPGA: DE2-115

7-Segment Display 1

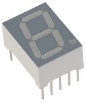
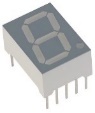
7-Segment Display 5

7-Segment Display 4

7-Segment Display 3

7-Segment Display 2

7-Segment Display 1



ROM

4-16 Decoder

4-16 Decoder

Address Data

4-bit Adder

Timeout

Clock

Reset

Clock

Reset

Digit Timer

Load Register Player 2

Clock

Reset

Random Number Generator

B1

B2

Enable

B3

Player

B2

Enable

B1

B3

Green LED

Red LED

Address

Data

Access Controller

Clock

Reset

B3

B2

Button Shaper

Clock

Reset

Button Shaper

Timeout

B1

B2

B3

B1

Password

Password

Load P1

Player 1

Reset

Load P2

Password

Player 1

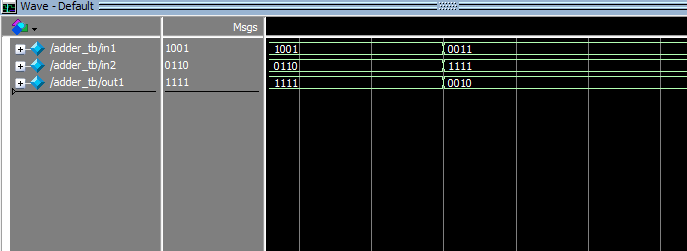
Player 2

**Figure 1.** Top-level system architecture for the binary mental math game.

# Simulation Results

4-bit Adder Test:

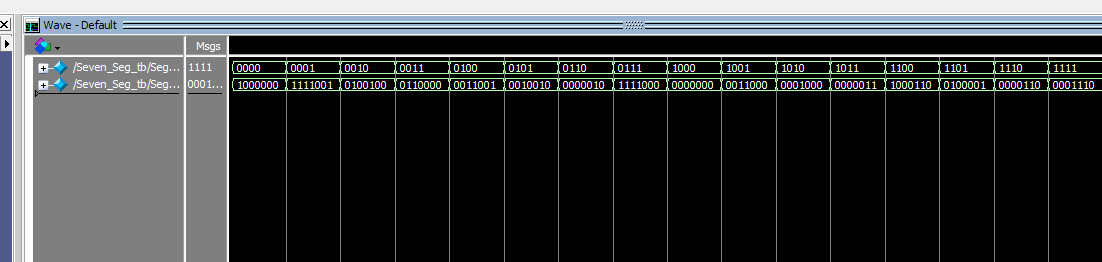
Figure 2 shows the simulation done to test the adder Verilog file. Results show that the adder is working as intended.



**Figure 2.** 4-Bit Adder Simulation. Using two sets of inputs, this simulation shows the respective outputs. By doing binary addition, we can see that the output shows the correct sum for both tests.

4-16 decoder: 7-segment display:

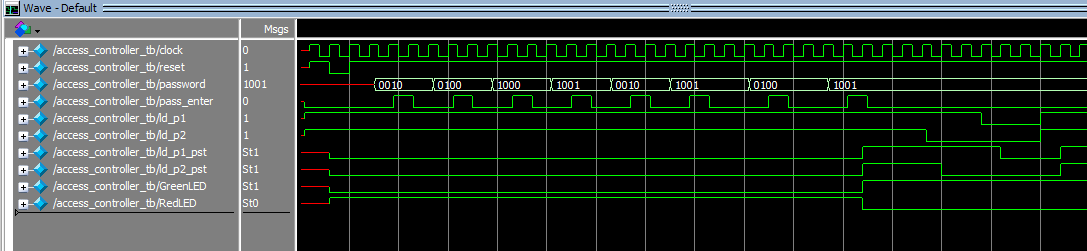
Figure 3 shows the simulation done to test the 7-segment display. The LEDs on the 7-segment display are active-low meaning an output of 0 for a bit enables that bit.



**Figure 3.** 4-16 Decoder Simulation. This simulation shows all possible outputs using the full range of a 4-bit input.

Access Controller:

Figure 4 shows the access controller simulation. Three tests were conducted to check the viability of the code. The first test was an incorrect password input. The second test was a correct password input. The final test was to see if outputs would change when inputs were changed.



**Figure 4.** Access Controller Testbench Simulation.

Button shaper:

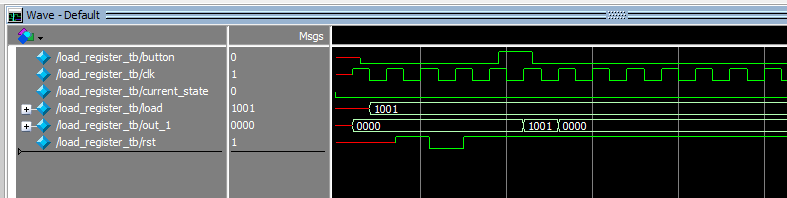
Figure 5 shows the simulation done for the button shaper module.

A screen shot of a clock

Description automatically generated

**Figure 5.** Button Shaper Testbench Simulation. Note that b\_out only remains in the active state for one clock cycle regardless of the length of b\_in.

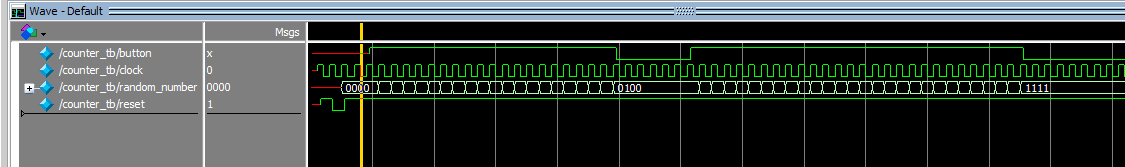
Figure 6 shows the simulation results for the load register. The output remains 0 until the button is pressed, at which point the output changes to the value that the player entered.

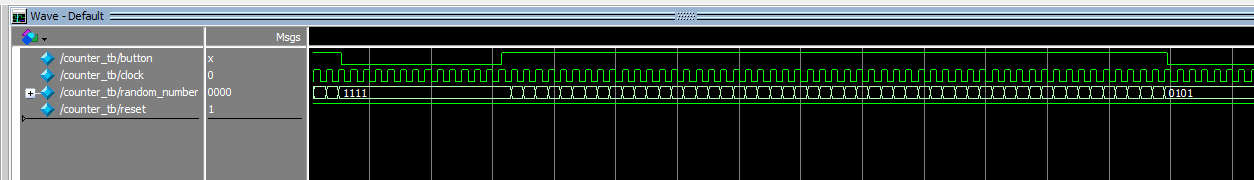


**Figure 6.** Load Register Testbench Simulation.

True Random Number Generator:

Figure 7 shows the simulation results for the true random number generator. It functions by cycling through 0-F as a 4-bit number while the button is pressed. Once the button is released, the last value that the program cycled to is sent to be displayed.

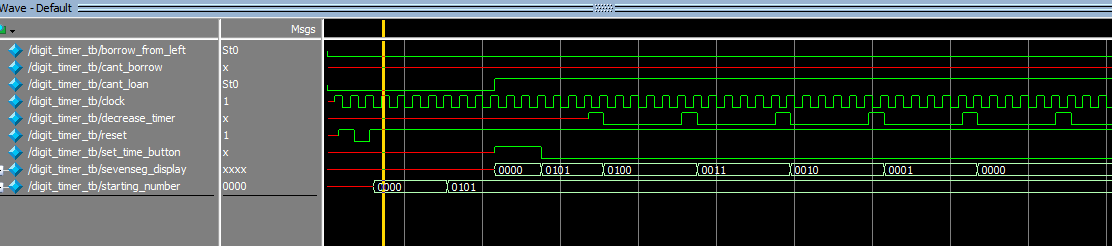




**Figure 7.** Testbench Simulation of True Random Number Generator. Each test was conducted over a period of clock edges greater than 16. As evidenced, the output random\_number wraps around back to 0000.

Digit Timer:

Figure 8 shows the simulation results for a single digit timer module. The digit timer is controlled by a 1 second timer. It is designed to be modular and it is possible to attach as many digit timer modules together as desired.



**Figure 8.** Testbench simulation of a digit timer. After

# FPGA Board Testing Results

The following images in figures 8-11 show the DE2-115 FPGA board with the Quartus file uploaded to it.

A circuit board

Description automatically generated

**Figure 8.** Reset State. In this figure, the reset button has been pressed. All values have been zeroed out and players have been logged out of the board.

A circuit board

Description automatically generated

**Figure 9.** Access Granted. This image shows the green LED enabled after the user has entered the correct password using switches 10-7.

A circuit board

Description automatically generated

**Figure 10.** Player 1 input. This image shows the 7-segment displays after player 1 has used switches 17-14 to input a 4-bit value and then pressed the load button.

A circuit board

Description automatically generated

**Figure 11.** Player 2 input. Here, player 2 has used switches 3-0 to enter a 4-bit number and pressed their load button. The sum has also been computed and displayed.

The password control and load registers function in tandem with the button pusher modules to control player input. We can see through these pictures that the 4-bit binary adder is working and that inputs are properly being converted from their 4-bit form to their hexadecimal equivalent.

# Video Demo

The following links are to videos showing the operation of binary mental math game. For each lab.

Lab 1:

<https://drive.google.com/file/d/1CQv7KU_kPwXUhqRWP6rJahYrf8mgxPv_/view?usp=sharing>

Lab 2:

<https://drive.google.com/file/d/1FHTdyxH9T4_aaNNfueHTf-uf1d4xCvPb/view?usp=sharing>

Lab 3:

//Video not available

# Conclusion

Lab 4 had us implement ROM into the password portion of the access controller. The game is a 1-player game that can be used to practice and test a person’s ability to convert numbers between hexadecimal and binary. Additionally, the game tests a person’s ability to add in binary. The goal of the game is for the player to determine which binary number must be added to the randomly generated input in order to reach binary 1111. The additions made in lab 4 helped teach about ROM and how it works and also made the password control part of the access controller more modular.

# Appendix

The following code includes all 3 files for Lab 4.

**Lab4\_KHAN\_M**

module Lab4\_KHAN\_M (reset, clock, password\_button, player\_button, password, red\_LED, green\_LED, player\_input,seg7\_num1, seg7\_num2, seg7\_sum, tens\_time, ones\_time, seg7\_tens, seg7\_ones);

//lab 4 variables

wire [15:0] data;

//lab 3 variables

input [3:0] tens\_time, ones\_time; //two digit time connections

wire enable, reconfig, timeout, second\_count; // two digit timer connections

wire [3:0] random\_number, sevenseg\_display\_tens, sevenseg\_display\_ones;

output [6:0] seg7\_tens, seg7\_ones;

//lab 2 variables

input clock, reset, password\_button, player\_button;

input [3:0] password;

output red\_LED, green\_LED;

wire random\_number\_button, player\_button\_pressed, password\_button\_pressed, ac\_to\_random, ac\_to\_ld\_reg;

wire [3:0] output\_from\_ld\_reg\_p2;

//lab 1 variables

input [3:0] player\_input;

output [6:0] seg7\_num1, seg7\_num2, seg7\_sum;

wire [3:0] sum;

//lab 4 function calls

rom\_controller rom\_controller\_instance(reset, password\_button\_pressed, clock, data);

//lab 3 function calls

random\_number\_generator rng(clock, reset, ac\_to\_random, random\_number);

onesecond\_timer second\_decrement(reset, clock, enable, second\_count);

two\_digit\_timer timedisplay(clock, reset, reconfig, tens\_time, ones\_time, second\_count, timeout, sevenseg\_display\_tens, sevenseg\_display\_ones);

Seven\_Seg tens\_display (sevenseg\_display\_tens, seg7\_tens);

Seven\_Seg ones\_display (sevenseg\_display\_ones, seg7\_ones);

//lab 2 function calls

button\_shaper player\_button\_shaper(reset, clock, player\_button, player\_button\_pressed);

button\_shaper password\_button\_shaper(reset, clock, password\_button, password\_button\_pressed);

load\_register player\_load (reset, clock, player\_input, ac\_to\_ld\_reg, output\_from\_ld\_reg\_p2);

access\_controller game\_access (reset, clock, random\_number\_button, player\_button\_pressed, password, password\_button\_pressed, ac\_to\_random, ac\_to\_ld\_reg, red\_LED, green\_LED, enable, reconfig, timeout, data);

//lab 1 function calls

adder adder\_1 (random\_number, output\_from\_ld\_reg\_p2, sum);

Seven\_Seg seg7\_number1 (ac\_to\_random, seg7\_num1);

Seven\_Seg seg7\_number2 (output\_from\_ld\_reg\_p2, seg7\_num2);

Seven\_Seg seg7\_summation (sum, seg7\_sum);

Endmodule

**Rom\_controller**

module ROM\_controller (reset, inc, clock, data);

input reset, inc, clock;

output [15:0] data;

reg [4:0] addr;

ROM\_1 ROM\_1\_instance(addr, clock, data);

always @(posedge clock) begin

if (reset == 0)

addr <= 0;

else if (inc == 1)

addr <= addr + 1;

end

endmodule

**ROM Testbench**

`timescale 10ns/ 100ps

module ROM\_SIM\_tb();

reg[4:0] addr;

reg clk;

wire[15:0] data;

ROM\_1 ROM\_1\_DUT(addr, clk, data);

always begin

clk = 0;

#5; clk <= 1;

#5;

end

initial begin

addr <= 0;

#55;

@(posedge clk);

addr <= 1;

@(posedge clk);

addr <= 2;

@(posedge clk);

addr <= 3;

@(posedge clk);

addr <= 4;

@(posedge clk);

addr <= 5;

@(posedge clk);

addr <= 6;

@(posedge clk);

addr <= 7;

@(posedge clk);

addr <= 8;

@(posedge clk);

addr <= 9;

@(posedge clk);

@(posedge clk);

@(posedge clk);

@(posedge clk);

@(posedge clk);

addr <= 30;

@(posedge clk);

addr <= 31;

end

endmodule

**ROM\_1**

// megafunction wizard: %ROM: 1-PORT%

// GENERATION: STANDARD

// VERSION: WM1.0

// MODULE: altsyncram

// ============================================================

// File Name: ROM\_1.v

// Megafunction Name(s):

// altsyncram

//

// Simulation Library Files(s):

// altera\_mf

// ============================================================

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

//

// 18.1.0 Build 625 09/12/2018 SJ Lite Edition

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Copyright (C) 2018 Intel Corporation. All rights reserved.

//Your use of Intel Corporation's design tools, logic functions

//and other software and tools, and its AMPP partner logic

//functions, and any output files from any of the foregoing

//(including device programming or simulation files), and any

//associated documentation or information are expressly subject

//to the terms and conditions of the Intel Program License

//Subscription Agreement, the Intel Quartus Prime License Agreement,

//the Intel FPGA IP License Agreement, or other applicable license

//agreement, including, without limitation, that your use is for

//the sole purpose of programming logic devices manufactured by

//Intel and sold by Intel or its authorized distributors. Please

//refer to the applicable agreement for further details.

// synopsys translate\_off

`timescale 1 ps / 1 ps

// synopsys translate\_on

module ROM\_1 (

address,

clock,

q);

input [4:0] address;

input clock;

output [15:0] q;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_off

`endif

tri1 clock;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_on

`endif

wire [15:0] sub\_wire0;

wire [15:0] q = sub\_wire0[15:0];

altsyncram altsyncram\_component (

.address\_a (address),

.clock0 (clock),

.q\_a (sub\_wire0),

.aclr0 (1'b0),

.aclr1 (1'b0),

.address\_b (1'b1),

.addressstall\_a (1'b0),

.addressstall\_b (1'b0),

.byteena\_a (1'b1),

.byteena\_b (1'b1),

.clock1 (1'b1),

.clocken0 (1'b1),

.clocken1 (1'b1),

.clocken2 (1'b1),

.clocken3 (1'b1),

.data\_a ({16{1'b1}}),

.data\_b (1'b1),

.eccstatus (),

.q\_b (),

.rden\_a (1'b1),

.rden\_b (1'b1),

.wren\_a (1'b0),

.wren\_b (1'b0));

defparam

altsyncram\_component.address\_aclr\_a = "NONE",

altsyncram\_component.clock\_enable\_input\_a = "BYPASS",

altsyncram\_component.clock\_enable\_output\_a = "BYPASS",

altsyncram\_component.init\_file = "ROM\_1.mif",

altsyncram\_component.intended\_device\_family = "Cyclone IV E",

altsyncram\_component.lpm\_hint = "ENABLE\_RUNTIME\_MOD=NO",

altsyncram\_component.lpm\_type = "altsyncram",

altsyncram\_component.numwords\_a = 32,

altsyncram\_component.operation\_mode = "ROM",

altsyncram\_component.outdata\_aclr\_a = "NONE",

altsyncram\_component.outdata\_reg\_a = "CLOCK0",

altsyncram\_component.widthad\_a = 5,

altsyncram\_component.width\_a = 16,

altsyncram\_component.width\_byteena\_a = 1;

endmodule

// ============================================================

// CNX file retrieval info

// ============================================================

// Retrieval info: PRIVATE: ADDRESSSTALL\_A NUMERIC "0"

// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"

// Retrieval info: PRIVATE: AclrByte NUMERIC "0"

// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_ENABLE NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_SIZE NUMERIC "8"

// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_INPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_OUTPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: Clken NUMERIC "0"

// Retrieval info: PRIVATE: IMPLEMENT\_IN\_LES NUMERIC "0"

// Retrieval info: PRIVATE: INIT\_FILE\_LAYOUT STRING "PORT\_A"

// Retrieval info: PRIVATE: INIT\_TO\_SIM\_X NUMERIC "0"

// Retrieval info: PRIVATE: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: PRIVATE: JTAG\_ENABLED NUMERIC "0"

// Retrieval info: PRIVATE: JTAG\_ID STRING "NONE"

// Retrieval info: PRIVATE: MAXIMUM\_DEPTH NUMERIC "0"

// Retrieval info: PRIVATE: MIFfilename STRING "ROM\_1.mif"

// Retrieval info: PRIVATE: NUMWORDS\_A NUMERIC "32"

// Retrieval info: PRIVATE: RAM\_BLOCK\_TYPE NUMERIC "0"

// Retrieval info: PRIVATE: RegAddr NUMERIC "1"

// Retrieval info: PRIVATE: RegOutput NUMERIC "1"

// Retrieval info: PRIVATE: SYNTH\_WRAPPER\_GEN\_POSTFIX STRING "0"

// Retrieval info: PRIVATE: SingleClock NUMERIC "1"

// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"

// Retrieval info: PRIVATE: WidthAddr NUMERIC "5"

// Retrieval info: PRIVATE: WidthData NUMERIC "16"

// Retrieval info: PRIVATE: rden NUMERIC "0"

// Retrieval info: LIBRARY: altera\_mf altera\_mf.altera\_mf\_components.all

// Retrieval info: CONSTANT: ADDRESS\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_INPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_OUTPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: INIT\_FILE STRING "ROM\_1.mif"

// Retrieval info: CONSTANT: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: CONSTANT: LPM\_HINT STRING "ENABLE\_RUNTIME\_MOD=NO"

// Retrieval info: CONSTANT: LPM\_TYPE STRING "altsyncram"

// Retrieval info: CONSTANT: NUMWORDS\_A NUMERIC "32"

// Retrieval info: CONSTANT: OPERATION\_MODE STRING "ROM"

// Retrieval info: CONSTANT: OUTDATA\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: OUTDATA\_REG\_A STRING "CLOCK0"

// Retrieval info: CONSTANT: WIDTHAD\_A NUMERIC "5"

// Retrieval info: CONSTANT: WIDTH\_A NUMERIC "16"

// Retrieval info: CONSTANT: WIDTH\_BYTEENA\_A NUMERIC "1"

// Retrieval info: USED\_PORT: address 0 0 5 0 INPUT NODEFVAL "address[4..0]"

// Retrieval info: USED\_PORT: clock 0 0 0 0 INPUT VCC "clock"

// Retrieval info: USED\_PORT: q 0 0 16 0 OUTPUT NODEFVAL "q[15..0]"

// Retrieval info: CONNECT: @address\_a 0 0 5 0 address 0 0 5 0

// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0

// Retrieval info: CONNECT: q 0 0 16 0 @q\_a 0 0 16 0

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1.v TRUE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1.inc FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1.cmp FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1.bsf FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1\_inst.v FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ROM\_1\_bb.v TRUE

// Retrieval info: LIB\_FILE: altera\_mf

**Digit Timer:**

module digit\_timer (clock, reset, set\_time\_button, starting\_number, sevenseg\_display, cant\_loan, borrow\_from\_left, cant\_borrow, decrease\_timer);

input clock, reset, set\_time\_button, cant\_borrow, decrease\_timer;

input [3:0] starting\_number;

reg cant\_borrow\_var;

reg [3:0] digit\_count = 4'b0000;

output reg [3:0] sevenseg\_display;

output reg cant\_loan = 1'b0, borrow\_from\_left = 1'b0;

always @ (set\_time\_button, decrease\_timer) begin //when button is pressed, or when a second has passed do stuff

if (reset == 1'b0) begin //clear display

sevenseg\_display <= 4'b0000;

cant\_loan <= 1'b0;

borrow\_from\_left <= 1'b0;

cant\_borrow\_var <= cant\_borrow;

end

else begin

if (set\_time\_button == 1'b1) begin

if(starting\_number > 4'b1001) //if starting number is > 9, set timer value to 9.

digit\_count <= 4'b1001;

else

digit\_count <= starting\_number;

end

if(cant\_borrow\_var != 0) //for the leftmost timer

cant\_borrow\_var <= 1;

if(digit\_count == 4'b0001) begin

if( cant\_borrow\_var == 1'b1) begin

//digit\_count <= 4'b0000; //decrement to 0

cant\_loan <= 1'b1; // send signal to right to let it know this one is now 0 and can't loan

end

//else

//digit\_count <= digit\_count - 4'b0001;

end

if(digit\_count == 4'b0000) begin

if( cant\_borrow\_var == 1'b0) begin//look to see if the digit to the left can be borrowed from

borrow\_from\_left <= 1'b1; //send a signal to let the left know to decrement

digit\_count <= 4'b1001;

end

else //if there is nothing to borrow, then we are out of time

cant\_loan <= 1'b1;

end

else begin

if(decrease\_timer == 1'b1) //to ensure that the digit timer only decrements when the signal is set to 1 and not when it is changed back to 0

digit\_count <= digit\_count - 4'b0001;

end

end

sevenseg\_display <= digit\_count;

borrow\_from\_left <= 1'b0;

end

always @(posedge clock) begin // keep the value up to date

end

endmodule

**Digit Timer Testbench:**

`timescale 10ns/100ps

module digit\_timer\_tb ();

reg clock, reset, set\_time\_button, decrease\_timer, cant\_borrow;

reg [3:0] starting\_number;

wire [3:0] sevenseg\_display;

wire cant\_loan, borrow\_from\_left;

digit\_timer DUT\_digit\_timer (clock, reset, set\_time\_button, starting\_number, sevenseg\_display, cant\_loan, borrow\_from\_left, cant\_borrow, decrease\_timer);

always begin

#10 clock = 1;

#10 clock = 0;

end

initial begin

@(posedge clock);

#5 reset = 1;

@(posedge clock);

#5 reset = 0;

@(posedge clock);

#5 reset = 1;

#5 starting\_number = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 starting\_number = 4'b0101;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 set\_time\_button = 1;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 set\_time\_button = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1;

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1; //Time 5

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1;

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1;

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1;

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 decrease\_timer = 1; //0

@(posedge clock);

#5 decrease\_timer = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

end

endmodule

**True Random Number Generator:**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// random\_number\_generator

// This module generates a random number between 0 and 15

// It uses a counter that increments while the button is pressed to achieve a true random number.

//

module random\_number\_generator (clock, reset, button, random\_number);

input clock, reset, button;

output [3:0] random\_number;

assign inverted\_button\_signal = ~button;

counter random\_number\_counter(clock, reset, inverted\_button\_signal, random\_number);

endmodule

**True Random Number Generator Testbench:**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// random\_number\_generator\_tb

// This module generates a random number between 0 and 15

// It uses a counter that increments while the button is pressed to achieve a true random number.

// The button press is simulated by using a random number posedge clocks as a delay

`timescale 10ns/100ps

module random\_number\_generator\_tb ();

reg clock, reset, button;

wire random\_number;

random\_number\_generator DUT\_random\_number\_generator (clock, reset, button, random\_number);

always begin

#10 clock = 1;

#10 clock = 0;

end

initial begin

@(posedge clock);

#5 reset = 1;

@(posedge clock);

#5 reset = 0;

@(posedge clock);

#5 reset = 1;

//20

@(posedge clock);

@(posedge clock);

#5 button = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 button = 1;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

//27

#5 button = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

end

endmodule

**Counter**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// counter

// This module is the code 4-bit counter

// It simply adds one to the count at every posedge clock (loops at 15 because output is 4-bit)

//

module counter (clock, reset, button, random\_number);

input clock, reset, button;

output reg [3:0] random\_number;

always @(posedge clock) begin

if(reset == 1'b0)

random\_number <= 4'b0000;

else

if(button == 1'b1)

random\_number <= random\_number + 1;

end

endmodule

**Counter Testbench:**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// counter\_tb

// This module is the code 4-bit counter

// It simply adds one to the count at every posedge clock (loops at 15 because output is 4-bit)

//

`timescale 10ns/100ps

module counter\_tb ();

reg clock, reset, button;

wire [3:0] random\_number;

counter DUT\_counter (clock, reset, button, random\_number);

always begin

#10 clock = 1;

#10 clock = 0;

end

initial begin

@(posedge clock);

#5 reset = 1;

@(posedge clock);

#5 reset = 0;

@(posedge clock);

#5 reset = 1;

//20 positive clock edges output 10

@(posedge clock);

@(posedge clock);

#5 button = 1;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 button = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

//27 postive clock edges output 13

#5 button = 1;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 button = 0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

//54 postive clock edge output 11

#5 button = 1;

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

@(posedge clock);

#5 button = 0;

end

endmodule

**One Second Timer:**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// onesecond\_timer

// This module controls the time for the digit\_timer

// This module increments time by 1 second

//

module onesecond\_timer(reset, clock, second\_count);

input clock, reset;

output [7:0] second\_count;

reg [15:0] millisecond\_count = 0;

reg [7:0] second\_count = 0;

always @ (posedge clock) begin//count each second

if (reset == 1'b0) begin

millisecond\_count = 0;

second\_count = 0;

end

if (millisecond\_count == 26'd50000) begin //26'd50000

second\_count <= second\_count + 8'b0000\_0001;

millisecond\_count <= 0;

end

else begin

millisecond\_count <= millisecond\_count + 1'b1;

end

end

endmodule

**One Second Timer Testbench:**

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// onesecond\_timer

// This module controls the time for the digit\_timer

// This module increments time by 1 second

//

`timescale 10ns/100ps

module onesecond\_timer\_tb();

reg clock, reset;

wire [7:0] second\_count;

onesecond\_timer DUT\_onesecond\_timer(reset, clock, second\_count);

always begin

#10 clock = 1;

#10 clock = 0;

end

initial begin

reset = 1;

#10

reset = 0;

#10

reset = 1;

@(posedge clock);

end

endmodule

The following code includes all 7 files used for Lab 2.

**Access Controller:**

// ECE 5440

// Author: Mubashar Khan, 2949

// access\_controller

// This module is used to prevent access to the game until users have entered the correct password

// The program waits for players to enter a 4-digit password that has been predefined.

// The password that players should enter is 2949.

//

module access\_controller(reset, clock, random\_button, player\_input\_button, password, pass\_enter, output\_to\_rng, output\_to\_load\_reg, RedLED, GreenLED, enable, reconfig, timeout);

input reset, clock, random\_button, player\_input\_button, pass\_enter, timeout;

input [3:0] password;

output output\_to\_rng, output\_to\_load\_reg, RedLED, GreenLED, enable, reconfig;

reg password\_correct\_flag;

reg output\_to\_rng, output\_to\_load\_reg, RedLED, GreenLED, enable, reconfig;

reg [3:0] state;

parameter [3:0] s1\_first\_digit\_check = 4'b0000;

parameter [3:0] s2\_second\_digit\_check = 4'b0001;

parameter [3:0] s3\_third\_digit\_check = 4'b0010;

parameter [3:0] s4\_fourth\_digit\_check = 4'b0011;

parameter [3:0] s5\_success = 4'b0100; // state 4 will also check for successful password entry

parameter [3:0] s6\_reconfig = 4'b0101;

parameter [3:0] s7\_gamestart = 4'b0110;

parameter [3:0] s8\_gameplay = 4'b0111;

parameter [3:0] s9\_gameover = 4'b1000;

always @(posedge clock)

begin

//reg [1:0] password\_correct\_flag;

if (reset == 1'b0)

begin //reset all output variables to default values, goto state 1

state <= s1\_first\_digit\_check;

output\_to\_rng <= 1'b1; //1 because button signal is not inverted yet. button not pushed = 1, pushed = 0

output\_to\_load\_reg <= 1'b0;

GreenLED <= 1'b1;

RedLED <= 1'b1;

password\_correct\_flag <= 1'b1; //reset flag

end

else begin

case (state)

//State 1

s1\_first\_digit\_check:begin

output\_to\_rng <= 1'b1; // signal for random number generator

output\_to\_load\_reg <= 1'b0;

GreenLED <= 1'b0;

RedLED <= 1'b1;

password\_correct\_flag <= 1'b1;

if (pass\_enter == 1'b1)

begin

if (password == 4'b0010)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s2\_second\_digit\_check; // goto state 2 for second input

end

else begin

state <= s1\_first\_digit\_check; //repeat s1 until first digit has been entered

end

end

//State 2

s2\_second\_digit\_check:begin

//no change to outputs

if (pass\_enter == 1'b1)

begin

if (password == 4'b1001)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s3\_third\_digit\_check; // goto state 3 for third input

end

else begin

state <= s2\_second\_digit\_check; //repeat s2 until second digit has been entered

end

end

//State 3

s3\_third\_digit\_check:begin

if (pass\_enter == 1'b1)

begin

if (password == 4'b0100)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s4\_fourth\_digit\_check; // goto state 4 for fourth input

end

else begin

state <= s3\_third\_digit\_check; //repeat s3 until first digit has been entered

end

end

//State 4

s4\_fourth\_digit\_check: begin

if (pass\_enter == 1'b1)

begin

if (password == 4'b1001)

begin

if(password\_correct\_flag == 1'b1) //password correct

begin

//output\_to\_rng <= input\_button\_press\_p1;

//output\_to\_load\_reg <= player\_input\_button;

RedLED <= 1'b0;

GreenLED <= 1'b1;

state <= s5\_success; // goto state 5 for infinite success loop

end

end

else begin

password\_correct\_flag <= 1'b0;

end

if(password\_correct\_flag == 1'b0)

begin

state <= s1\_first\_digit\_check; // goto state 1 for first input if password is incorrect

end

end

else begin

state <= s4\_fourth\_digit\_check; //repeat s4 until first digit has been entered

end

end

//State 5

s5\_success: begin

//output\_to\_rng <= input\_button\_press\_p1; //random number

//output\_to\_load\_reg <= player\_input\_button;

RedLED <= 1'b0;

GreenLED <= 1'b1;

state <= s6\_reconfig;

end

s6\_reconfig: begin

if(pass\_enter == 1'b1) begin

reconfig <= 1'b1;

state <= s7\_gamestart;

end

else

state <= s6\_reconfig;

end

s7\_gamestart: begin

reconfig <= 1'b0;

if(pass\_enter == 1'b1) begin

enable <= 1'b1;

state <= s8\_gameplay;

end

else

state <= s7\_gamestart;

end

s8\_gameplay: begin

output\_to\_rng <= random\_button; //random number

output\_to\_load\_reg <= player\_input\_button;

if (timeout == 1'b1) begin

state <= s9\_gameover;

end

else

state <= s8\_gameplay;

end

s9\_gameover: begin

enable <= 1'b0;

RedLED <= 1'b1;

GreenLED <= 1'b0;

state <= s9\_gameover;

end

//default

default: begin

state <= s1\_first\_digit\_check;

end

endcase

end

end

endmodule

**Access Controller Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// access\_controller\_tb

// This module is used to test access\_controller.v

// The test cycles through the 4 states using a incorrect and correct password sequence to test if the module works properly

//

`timescale 1 ns/100 ps

module access\_controller\_tb ();

reg [0:0] reset, clock, ld\_p1, ld\_p2, pass\_enter;

reg [3:0] password;

wire [0:0] ld\_p1\_pst, ld\_p2\_pst; // 1 on success only

wire [0:0] RedLED, GreenLED; //on/off

access\_controller DUT\_access\_controller (reset, clock, ld\_p1, ld\_p2, password, pass\_enter, ld\_p1\_pst, ld\_p2\_pst, RedLED, GreenLED);

always begin //enable clock

#10 clock = 1'b1;

#10 clock = 1'b0;

end

initial begin //testing

#5

ld\_p1 = 1'b1;

ld\_p2 = 1'b1;

pass\_enter = 1'b0;

@(posedge clock);

reset = 1'b1;

@(posedge clock);

reset = 1'b0;

@(posedge clock);

reset = 1'b1;

//password is 2949

//incorrect input test

@(posedge clock);

#5 password = 4'b0010; // 2

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b0100; // 4 <-should trigger password\_correct\_flag

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1000; // 8

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

//correct input test

//@(posedge clock);

//#5 reset = 0;

@(posedge clock);

#5 password = 4'b0010; // 2

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

@(posedge clock);

#5 password = 4'b0100; // 4

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

// now testing output if inputs are changed while in state 5

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b1;

ld\_p2 = 1'b0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b0;

ld\_p2 = 1'b0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b1;

ld\_p2 = 1'b1;

//reset everything

@(posedge clock);

reset = 0;

@(posedge clock);

@(posedge clock);

reset = 1;

end

endmodule

**Load Register:**

// ECE 5440

// Author: Mubashar Khan, 2949

// load\_register

// This module is used to prevent inputs from showing up on the displays until the load button has been pressed

// The program waits for players to enter a 4-bit input and press the load button.

// The signal then allows the input to be displayed.

//

module load\_register(rst, clk, four\_bit\_input, button, four\_bit\_output);

input [3:0] four\_bit\_input;

input rst, clk, button;

output reg [3:0] four\_bit\_output;

always @(posedge clk) begin

if (rst == 1'b0) begin

four\_bit\_output <= 4'b0000;

end

else begin

if (button == 1'b1) begin

four\_bit\_output <= four\_bit\_input;

end

end

end

endmodule

**Load Register Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// load\_register\_tb

// This module is used to test the load register

// The program waits for players to press the load button.

//

`timescale 1 ns/100 ps

module load\_register\_tb();

reg clk, rst, button;

reg [3:0]load;

reg current\_state = 2'b00;

wire [3:0] out\_1;

load\_register DUT\_load\_register(rst, clk, load, button, out\_1);

always begin

#10 clk = 1;

#10 clk = 0;

end

initial begin

@(posedge clk);

#5 button = 0;

#5 load = 4'b1001;

@(posedge clk);

#5 rst = 1;

@(posedge clk);

#5 rst = 0;

@(posedge clk);

#5 rst = 1;

@(posedge clk);

#5 button = 1;

@(posedge clk);

#5 button = 0;

end

endmodule

**Button Shaper:**

// ECE 5440

// Author: Mubashar Khan, 2949

// button\_shaper

// This module is a button shaper that emits a pulse for 1 clock cycle when the button is pressed

// This module was created using a two procedure fsm

//

module button\_shaper(reset, clk, button\_shaper\_input, button\_shaper\_output);

input clk, reset, button\_shaper\_input;

output reg button\_shaper\_output;

reg [1:0] current\_state;

reg [1:0] next\_state;

parameter [1:0] init\_state = 2'b00;

parameter [1:0] pulse\_state = 2'b01;

parameter [1:0] wait\_state = 2'b10;

always @ (current\_state, button\_shaper\_input)

begin

case(current\_state)

init\_state:

begin

button\_shaper\_output <= 1'b0;

if (button\_shaper\_input == 1'b0)

next\_state <= pulse\_state;

else

next\_state <= init\_state;

end

pulse\_state:

begin

button\_shaper\_output <= 1'b1;

next\_state <= wait\_state;

end

wait\_state:

begin

button\_shaper\_output <= 1'b0;

if (button\_shaper\_input == 1'b1)

next\_state <= init\_state;

else

next\_state <= wait\_state;

end

default:

begin

button\_shaper\_output <= 1'b0;

next\_state <= init\_state;

end

endcase

end

always @(posedge clk)

begin

if (reset == 1'b0)

current\_state <= init\_state;

else

current\_state <= next\_state;

end

endmodule

**Button Shaper Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// button\_shaper\_tb

// This module is a button shaper that emits a pulse for 1 clock cycle when the button is pressed

// This module was created using a two procedure fsm

//

`timescale 10ns/100ps

module button\_shaper\_tb ();

reg clk;

reg reset;

reg b\_in;

reg current\_state = 2'b00;

wire b\_out;

button\_shaper DUT\_button\_shaper\_1 (clk, reset, b\_in, b\_out);

always begin

#10 clk = 1;

#10 clk = 0;

end

initial

begin

//test 1

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 1;

//test 2

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 1;

end

endmodule

// Seven\_Seg

// This module is the code for the 4-16 decoder.

// Its function is to take a 4-bit input and convert it to a 7-segment number that can be displayed on a 7-segment display

//

module Seven\_Seg (Seg\_in, Seg\_out);

input [3:0] Seg\_in;

output [6:0] Seg\_out;

reg [6:0] Seg\_out;

always @ (Seg\_in)

begin

case(Seg\_in)

4'b0000 : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

4'b0001 : begin Seg\_out = 7'b1111001; end //1 //7'b1111001; end 7'b1001111; end

4'b0010 : begin Seg\_out = 7'b0100100; end //2 //7'b0100100; end 7'b0010010; end

4'b0011 : begin Seg\_out = 7'b0110000; end //3 //7'b0110000; end 7'b0000110; end

4'b0100 : begin Seg\_out = 7'b0011001; end //4 //7'b0011001; end 7'b1001100; end

4'b0101 : begin Seg\_out = 7'b0010010; end //5 //7'b0010010; end 7'b0100100; end

4'b0110 : begin Seg\_out = 7'b0000010; end //6 //7'b0000010; end 7'b0100000; end

4'b0111 : begin Seg\_out = 7'b1111000; end //7 //7'b1111000; end 7'b0001111; end

4'b1000 : begin Seg\_out = 7'b0000000; end //8 //7'b0000000; end 7'b0000000; end

4'b1001 : begin Seg\_out = 7'b0011000; end //9 //7'b0011000; end 7'b0001100; end

4'b1010 : begin Seg\_out = 7'b0001000; end //A //7'b0001000; end 7'b0001000; end

4'b1011 : begin Seg\_out = 7'b0000011; end //b //7'b0000011; end 7'b1100000; end

4'b1100 : begin Seg\_out = 7'b1000110; end //C //7'b1000110; end 7'b0110001; end

4'b1101 : begin Seg\_out = 7'b0100001; end //d //7'b0100001; end 7'b1000010; end

4'b1110 : begin Seg\_out = 7'b0000110; end //E //7'b0000110; end 7'b0110000; end

4'b1111 : begin Seg\_out = 7'b0001110; end //F //7'b0001110; end 7'b0111000; end

default : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

endcase

end

endmodule

// Seven\_Seg\_tb

// This module is the test bench for the 4-16 decoder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module Seven\_Seg\_tb ();

reg[3:0] Seg\_in;

wire[6:0] Seg\_out;

Seven\_Seg DUTSeven\_Seg(Seg\_in,Seg\_out);

initial

begin

Seg\_in = 4'b0000; // 0

#10;

Seg\_in = 4'b0001; // 1

#10;

Seg\_in = 4'b0010; // 2

#10;

Seg\_in = 4'b0011; // 3

#10;

Seg\_in = 4'b0100; // 4

#10;

Seg\_in = 4'b0101; // 5

#10;

Seg\_in = 4'b0110; // 6

#10;

Seg\_in = 4'b0111; // 7

#10;

Seg\_in = 4'b1000; // 8

#10;

Seg\_in = 4'b1001; // 9

#10;

Seg\_in = 4'b1010; // A

#10;

Seg\_in = 4'b1011; // b

#10;

Seg\_in = 4'b1100; // C

#10;

Seg\_in = 4'b1101; // d

#10;

Seg\_in = 4'b1110; // E

#10;

Seg\_in = 4'b1111; // F

end

endmodule

// adder

// This module adds 2 four-bit inputs. We do not care about overflow and the output is also four-bits.

//

module adder (in1, in2, sum);

input [3:0] in1, in2;

output [3:0] sum;

reg [3:0] sum;

always @ (in1, in2)

begin

sum = in1 + in2;

end

endmodule

// adder\_tb

// This module is the test bench for the adder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module adder\_tb ();

reg [3:0] in1, in2;

wire [3:0] out1;

adder DUT\_adder(in1, in2, out1);

initial

begin

in1 = 4'b1001;

in2 = 4'b0110;

#10;

in1 = 4'b0011;

in2 = 4'b1111;

end

endmodule

The following is all the code created for Lab 1. It consists of 5 modules, Lab1\_KHAN\_M, Seven\_Seg.v, Seven\_Seg\_tb.v, adder.v, and adder\_tb.v.

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// Lab1\_KHAN\_M

// This module in the top code that calls all necessary functions

// Its function is to take thhe 4 bit input from players and pass those values to the adder and decoder functions.

// The returned value from the adder function is also passed to a decoder and all 3 decoder values are used to enable the proper pins on the 7-segment display.

//

module Lab1\_KHAN\_M (

//inputs

switch\_num1, switch\_num2,

//outputs

seg7\_num1, seg7\_num2, seg7\_sum);

input [3:0] switch\_num1, switch\_num2;

output [6:0] seg7\_num1, seg7\_num2, seg7\_sum;

wire [3:0] sum;

adder adder\_1 (switch\_num1, switch\_num2, sum);

Seven\_Seg seg7\_number1 (switch\_num1, seg7\_num1);

Seven\_Seg seg7\_number2 (switch\_num2, seg7\_num2);

Seven\_Seg seg7\_summation (sum, seg7\_sum);

Endmodule

// Seven\_Seg

// This module is the code for the 4-16 decoder.

// Its function is to take a 4-bit input and convert it to a 7-segment number that can be displayed on a 7-segment display

//

module Seven\_Seg (Seg\_in, Seg\_out);

input [3:0] Seg\_in;

output [6:0] Seg\_out;

reg [6:0] Seg\_out;

always @ (Seg\_in)

begin

case(Seg\_in)

4'b0000 : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

4'b0001 : begin Seg\_out = 7'b1111001; end //1 //7'b1111001; end 7'b1001111; end

4'b0010 : begin Seg\_out = 7'b0100100; end //2 //7'b0100100; end 7'b0010010; end

4'b0011 : begin Seg\_out = 7'b0110000; end //3 //7'b0110000; end 7'b0000110; end

4'b0100 : begin Seg\_out = 7'b0011001; end //4 //7'b0011001; end 7'b1001100; end

4'b0101 : begin Seg\_out = 7'b0010010; end //5 //7'b0010010; end 7'b0100100; end

4'b0110 : begin Seg\_out = 7'b0000010; end //6 //7'b0000010; end 7'b0100000; end

4'b0111 : begin Seg\_out = 7'b1111000; end //7 //7'b1111000; end 7'b0001111; end

4'b1000 : begin Seg\_out = 7'b0000000; end //8 //7'b0000000; end 7'b0000000; end

4'b1001 : begin Seg\_out = 7'b0011000; end //9 //7'b0011000; end 7'b0001100; end

4'b1010 : begin Seg\_out = 7'b0001000; end //A //7'b0001000; end 7'b0001000; end

4'b1011 : begin Seg\_out = 7'b0000011; end //b //7'b0000011; end 7'b1100000; end

4'b1100 : begin Seg\_out = 7'b1000110; end //C //7'b1000110; end 7'b0110001; end

4'b1101 : begin Seg\_out = 7'b0100001; end //d //7'b0100001; end 7'b1000010; end

4'b1110 : begin Seg\_out = 7'b0000110; end //E //7'b0000110; end 7'b0110000; end

4'b1111 : begin Seg\_out = 7'b0001110; end //F //7'b0001110; end 7'b0111000; end

default : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

endcase

end

endmodule

// Seven\_Seg\_tb

// This module is the test bench for the 4-16 decoder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module Seven\_Seg\_tb ();

reg[3:0] Seg\_in;

wire[6:0] Seg\_out;

Seven\_Seg DUTSeven\_Seg(Seg\_in,Seg\_out);

initial

begin

Seg\_in = 4'b0000; // 0

#10;

Seg\_in = 4'b0001; // 1

#10;

Seg\_in = 4'b0010; // 2

#10;

Seg\_in = 4'b0011; // 3

#10;

Seg\_in = 4'b0100; // 4

#10;

Seg\_in = 4'b0101; // 5

#10;

Seg\_in = 4'b0110; // 6

#10;

Seg\_in = 4'b0111; // 7

#10;

Seg\_in = 4'b1000; // 8

#10;

Seg\_in = 4'b1001; // 9

#10;

Seg\_in = 4'b1010; // A

#10;

Seg\_in = 4'b1011; // b

#10;

Seg\_in = 4'b1100; // C

#10;

Seg\_in = 4'b1101; // d

#10;

Seg\_in = 4'b1110; // E

#10;

Seg\_in = 4'b1111; // F

end

endmodule

// adder

// This module adds 2 four-bit inputs. We do not care about overflow and the output is also four-bits.

//

module adder (in1, in2, sum);

input [3:0] in1, in2;

output [3:0] sum;

reg [3:0] sum;

always @ (in1, in2)

begin

sum = in1 + in2;

end

endmodule

// adder\_tb

// This module is the test bench for the adder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module adder\_tb ();

reg [3:0] in1, in2;

wire [3:0] out1;

adder DUT\_adder(in1, in2, out1);

initial

begin

in1 = 4'b1001;

in2 = 4'b0110;

#10;

in1 = 4'b0011;

in2 = 4'b1111;

end

endmodule